Chapter 9. SOAP Interoperability and WSDL

All of the examples to this point have used the same SOAP implementation for both client and server. Our Apache SOAP server examples were accessed using Apache SOAP client applications, and our GLUE server examples were accessed by GLUE client applications. In those rare occasions when you have control over the technology used at every node of a distributed system, it's easiest to use the same technology throughout. However, that opportunity doesn't present itself all that often, and it's fundamentally at odds with the web services vision of the computing world. So it's necessary to investigate how SOAP implementations interoperate with each other.

There are dozens of SOAP implementations available right now, and others will be showing up every day. Over time, you'll probably find lots of existing enterprise systems making themselves accessible via a SOAP mechanism over a variety of transports. Some systems will undoubtedly use SOAP under the covers, so you won't have to deal with it at all. And new distributed software frameworks based on SOAP will certainly sprout up. How well can we expect these systems to interoperate with one another? After all, software is still developed largely by companies and individuals that are competing in one way or another, which usually leads to problems with interoperability. Sometimes the problems arise because the specification has one or more sections that are open to interpretation, and developers working for different organizations interpret things differently. The fact that these problems occur doesn't necessarily indicate that the specification is flawed; it only indicates that software development is ultimately a human enterprise, and humans are prone to disagree.

One frequent cause of interoperability problems is the `SOAPAction` field, which SOAP uses to send information to a recipient outside the bounds of the SOAP envelope. This field is carried as part of the underlying transport's header information; for example, when the SOAP message is carried over an HTTP transport, this field is sent as an HTTP header. (It's not clear how `SOAPAction` should be handled for transports other than HTTP.) The content of the field is a URI intended to provide information to the recipient. This information might be the service namespace, service method name, or something else. Whatever it is, it's intended to give the recipient some clue about the contents of the envelope, without requiring the recipient to parse the XML. Some SOAP implementations don't require any data value to be sent in the `SOAPAction` field, while others do. Knowing what is required and making sure to include it in the message is critical to SOAP interoperability. We'll look at the `SOAPAction` requirements of some current SOAP implementations and the APIs for setting its value in the examples in this chapter. But first, we'll look at the Web Services Definition Language (WSDL), an important standard that promises to help solve interoperability problems.

9.1 Web Services Definition Language

One way to avoid interoperability problems with SOAP-based services is to use a structured language to describe the service, its location, the service methods, parameters, data types, and so on. The Web Services Definition Language (WSDL) does just that. WSDL is an XML grammar for describing web services. Systems can determine the programmatic interface of a web service by looking at the WSDL document associated with that service. The document describes the service methods along with their parameters and return types, and may also include the address, or endpoint, of the service. One of the greatest benefits of WSDL is that it is a single, accepted standard[1] for describing web services, which among other things motivates SOAP developers to avoid using their own mechanism for that task.

Does WSDL eliminate the problems associated with human (or machine) interpretation? Well, not completely. It does mean that service descriptions are far less ambiguous than they would be if there were no standard, but on the other hand, WSDL has a specification of its own, and that specification has the potential to be interpreted differently by multiple parties. WSDL isn't a perfect solution, but it certainly puts those of us working with these technologies in a better place than we'd be in otherwise.

9.1.1 Overview of WSDL

We're not going to cover WSDL in detail, as it's a large subject that warrants a book of its own. But it's a major contributor toward achieving SOAP interoperability, so it's important to spend some time looking at it. Some SOAP implementations, like GLUE, incorporate WSDL as an integral part of their operation. GLUE examples throughout this book have used the `wsdl2java` utility to generate the client-side Java code for accessing a GLUE-based service. `wsdl2java` reads a WSDL document that describes a service in order to create the Java interface and class files. We didn't have to actually write the WSDL document in those examples because GLUE generates WSDL automatically. The GLUE client-side binding process also uses the WSDL for the service; that's how it creates the binding that is hidden behind the Java interface you use to interact with the service.

According to the WSDL specification:

> WSDL is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information. The operations and messages are described abstractly, and then bound to a concrete network protocol and message format to define an endpoint. Related concrete endpoints are combined into abstract endpoints (services).

A WSDL document is a collection of one or more service definitions. The document contains a root XML element named `definitions`; this element contains the service definitions. The `definitions` element can also contain an optional `targetNamespace` attribute, which specifies the URI associated with the service definitions. WSDL uses namespaces and namespace IDs in the same way we've been using them in SOAP envelopes, so it's common to find a number of namespaces declared at the `definitions` level of the document.

WSDL defines services as collections of network endpoints. These endpoints, in WSDL terminology, are known as `ports`. WSDL separates the service `ports` and their associated `messages` from the network protocol that the service is bound to (`binding`). The combination of a `binding` and a network address results in a `port`, and a `service` is a collection of those `ports`.

The WSDL specification defines the main document sections as follows:

[1] You can find the WSDL specification at http://www.w3.org/TR/wsdl.

`types`

A container for data type definitions using some type system (such as XSD)

`message`

An abstract, typed definition of the data being communicated

`operation`

An abstract description of an action supported by the service

`portType`

An abstract set of operations supported by one or more endpoints

`binding`

A concrete protocol and data format specification for a particular port type

`port`

A single endpoint defined as a combination of a binding and a network address

`service`

A collection of related endpoints

Let's take a look at a service that provides delayed stock quotes and news headlines based on a stock symbol. The service is called `urn:CorpDataServices`. It has a method called `getHeadlines` that takes a single `String` parameter for the stock symbol and returns a `String[]` that contains news headlines related to the company represented by the stock symbol. The service also contains a method named `getQuote` that takes the stock symbol as its parameter. The method returns an instance of a custom data type called `Quote` that contains the stock symbol, last trade price, change in price compared to its opening price, the number of shares of the stock traded so far today, and the timestamp of the information provided.

The following code implements the `Quote` custom type. I have placed this class in the `services` package. I'm separating the server-side classes from the client-side classes because I don't want to share the classes between client and server; there will be another package called `clients` for client-side classes. We'll use the techniques covered so far to map any custom types for use by client code.

```
package javasoap.book.ch9.services;
import java.io.*;
import java.util.*;
import java.net.*;
public class Quote {
   String _symbol;
   float  _lastPrice;
   float  _change;
   String _timeStamp;
   long   _volume;
   public String getSymbol(  ) {
      return _symbol;
   }
   public void setSymbol(String symbol) {
      _symbol = symbol;
   }
   public float getLastPrice(  ) {
      return _lastPrice;
   }
   public void setLastPrice(float lastPrice) {
      _lastPrice = lastPrice;
   }
   public float getChange(  ) {
      return _change;
   }
   public void setChange(float change) {
      _change = change;
   }
   public String getTimeStamp(  ) {
      return _timeStamp;
   }
   public void setTimeStamp(String timeStamp) {
      _timeStamp = timeStamp;
   }
   public long getVolume(  ) {
      return _volume;
   }
   public void setVolume(long volume) {
      _volume = volume;
   }
}
```

This is the only custom type we'll need for now; the rest of the data uses built-in types. Next we'll need a Java class that implements the service:

```
package javasoap.book.ch9.services;
import java.util.Vector;
public class CorpDataService {
   public Quote getQuote(String symbol)
            throws Exception {
      QuoteParser qp = new QuoteParser(  );
      return qp.getQuote(symbol);
   }
   public String[] getHeadlines(String symbol)
            throws Exception {
      HeadlineParser hp = new HeadlineParser(  );
      Vector v = hp.getHeadlines(symbol);
      int len = v.size(  );
      String[] result = new String[len];
      for (int i = 0; i < len; i++) {
         result[i] = (String)v.elementAt(i);
      }
      return result;
   }
}
```

The `getQuote( )` method of our `CorpDataService` class makes use of a class called `QuoteParser` . That class parses the data stream of a web-based quote service and packages the result in an instance of `javasoap.book.ch9.services.Quote`; however, I don't show the code for that class because it's specific to the quote system I'm using. I've made this service available at http://www.mindstrm.com/soap.htm so you won't need to run the service yourself. (I'll give you the information you need to access this service a little later.) Similarly, the `getHeadlines( )` method uses a class called `HeadlineParser` that collects news headlines as `String`s and returns them in an instance of `java.util.Vector`. `getHeadlines( )` creates a `String[]` based on the contents of the vector and returns the array to the caller.

I've decided to implement this service using GLUE. The service is located at *http://mindstrm.com:8004/glue/urn:CorpDataServices*.[2] Here's the `javasoap.book.ch9.services.CorpDataServices` class that I used to publish the service:

```
package javasoap.book.ch9.services;
import electric.util.Context;
import electric.server.http.HTTP;
import electric.registry.Registry;
public class CorpDataServices {

   public static void main( String[] args )
                  throws Exception {
      String ns = "urn:CorpDataServices";
      HTTP.startup("http://mindstrm.com:8004/glue");
      Context context = new Context(  );
      context.addProperty("activation", "application");
      context.addProperty("namespace", ns);
      context.addProperty("description", "Corporate Services Demo");
      Registry.publish(ns,
         Javasoap.book.ch9.services.CorpDataService.class, context);
   }
}
```

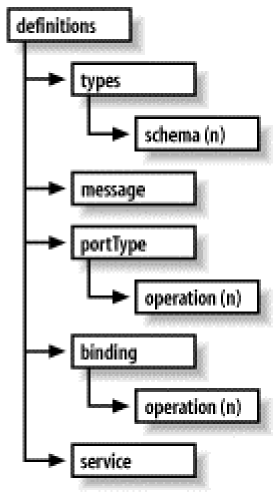
After you run the `CorpDataServices` example, you can request the WSDL document associated with the published services. We've done this with virtually every GLUE example. The binding process requests the WSDL, which is generated automatically by the server. Of course, you're free to generate your WSDL by hand; I prefer to let the system do it for me. The easiest way to get a look at the WSDL for a GLUE service is through your web browser. The WSDL for the `urn:CorpDataServices` service is located at *http://mindstrm.com:8004/glue/urn:CorpDataServices.wsdl*. Some browsers, like Internet Explorer, display the WSDL in a way that allows you to expand the various nodes to see their underlying contents. Regardless of which browser you're using, the WSDL returned by our service application is the same:

[2] Be sure to use *mindstrm.com*, not *www.mindstrm.com*. These URLs do not resolve to the same address.

```
<definitions name="CorpDataService"
  targetNamespace="http://www.themindelectric.com/wsdl/CorpDataService/"
  xmlns:tns="http://www.themindelectric.com/wsdl/CorpDataService/"
  xmlns:electric="http://www.themindelectric.com/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
  xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns="http://schemas.xmlsoap.org/wsdl/">
<types>
  <schema xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://www.themindelectric.com/package/java.lang/"
  targetNamespace="http://www.themindelectric.com/package/java.lang/">
    <complexType name="ArrayOfstring">
      <complexContent>
        <restriction base="soapenc:Array">
          <attribute ref="soapenc:arrayType" wsdl:arrayType="string[]" />
        </restriction>
      </complexContent>
    </complexType>
  </schema>
  <schema xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:tns=
     "http://www.themindelectric.com/package/javasoap.book.ch9.services/"
    targetNamespace=
     "http://www.themindelectric.com/package/javasoap.book.ch9.services/">
    <complexType name="Quote">
      <sequence>
        <element name="_symbol" nillable="true" type="string" />
        <element name="_lastPrice" type="float" />
        <element name="_change" type="float" />
        <element name="_timeStamp" nillable="true" type="string" />
        <element name="_volume" type="long" />
      </sequence>
    </complexType>
  </schema>
</types>
<message name="getQuote0SoapIn">
  <part name="arg0" type="xsd:string" />
</message>
<message name="getQuote0SoapOut">
  <part name="Result"
    xmlns:ns1=
      "http://www.themindelectric.com/package/javasoap.book.ch9.services/"
    type="ns1:Quote" />
</message>
<message name="getHeadlines1SoapIn">
  <part name="arg0" type="xsd:string" />
</message>
<message name="getHeadlines1SoapOut">
  <part name="Result"
    xmlns:ns1="http://www.themindelectric.com/package/java.lang/"
    type="ns1:ArrayOfstring" />
</message>
<portType name="CorpDataServiceSoap">
  <operation name="getQuote" parameterOrder="arg0">
    <input name="getQuote0SoapIn" message="tns:getQuote0SoapIn" />
    <output name="getQuote0SoapOut" message="tns:getQuote0SoapOut" />
  </operation>
  <operation name="getHeadlines" parameterOrder="arg0">
    <input name="getHeadlines1SoapIn"
           message="tns:getHeadlines1SoapIn" />
    <output name="getHeadlines1SoapOut"
           message="tns:getHeadlines1SoapOut" />
  </operation>
</portType>
<binding name="CorpDataServiceSoap" type="tns:CorpDataServiceSoap">
  <soap:binding style="rpc"
      transport="http://schemas.xmlsoap.org/soap/http" />
  <operation name="getQuote">
    <soap:operation soapAction="getQuote" style="rpc" />
    <input name="getQuote0SoapIn">
     <soap:body use="encoded" namespace="urn:CorpDataServices"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
    </input>
    <output name="getQuote0SoapOut">
     <soap:body use="encoded" namespace="urn:CorpDataServices"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
    </output>
  </operation>
  <operation name="getHeadlines">
   <soap:operation soapAction="getHeadlines" style="rpc" />
    <input name="getHeadlines1SoapIn">
     <soap:body use="encoded" namespace="urn:CorpDataServices"
       encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
    </input>
    <output name="getHeadlines1SoapOut">
     <soap:body use="encoded" namespace="urn:CorpDataServices"
       encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
    </output>
  </operation>
</binding>
<service name="CorpDataService">
  <documentation>Corporate Services Demo
  </documentation>
  <port name="CorpDataServiceSoap" binding="tns:CorpDataServiceSoap">
    <soap:address
      location="http://mindstrm.com:8004/glue/urn:CorpDataServices" />
  </port>
</service>
</definitions>
```

Let's walk through the WSDL generated for the `urn:CorpDataServices` service. There's quite a bit to look at, considering that we've got only two service methods and a single custom data type. (WSDL documents tend to be lengthy and hard to understand, even for extremely simple services. That's why you're best off letting a tool generate the WSDL for you.) Figure 9-1 illustrates the high-level structure of the WSDL document for our service. The "(n)" after some element names indicates that there is more than one element of a particular type.

Figure 9-1. High-level structure of a WSDL document



The `definitions` element is the root of the document; everything else lies within it. The `name` attribute contains the name of the service definitions contained within, which in this case is `CorpDataService`. Note that this is not the name of the service, `urn:CorpDataServices`, but the name given to the definitions of the service. This name is derived from the name of the Java class that implements the service, `javasoap.book.ch9.services.CorpDataService`. The `targetNamespace` attribute declares the namespace associated with the service definition. Its default name is created automatically by appending the name of the definition to `http://www.themindelectric.com/wsdl`.[3] The `definitions` element includes a number of namespace identifier declarations, all of which are pretty self-explanatory. The last attribute sets the overall namespace to `http://schemas.xmlsoap.org/wsdl`, which reflects the namespace for WSDL itself.

Next is the `types` element. This element defines the custom types used by the service. Each custom type is defined using a `schema` element, where each `schema` is a direct child of `types`. The first schema is used to define the string array returned by the `getHeadlines` service method. This string array is a `complexType`; remember that even an array is considered a complex type because it has multiple values. The `name` attribute is assigned the value `"ArrayOfstring"`. Inside `complexType`, we find an element named `complexContent`, which specifies that the type is restricted to an array of strings. The second `schema` element has a name attribute value of `"Quote"`. This is the custom type used to return stock quote data from the service, also defined using the `complexType` element. In this case the `complexType` contains a `sequence` element that, in turn, contains the data fields for the custom quote data type. Each field is defined using an `element` tag, and contains a `name` attribute that specifies the name of the data field. GLUE uses the actual member names from the `javasoap.book.ch9.services.Quote` class. Each element has its type specified as well.

The `message` elements come next. These elements define the information passed between client and server. Each `message` contains a `name` attribute, formed by starting with the service method name and adding a number to make sure that the name is unique. Finally, GLUE appends `SoapIn` to the name if the message is an `in` parameter, or `SoapOut` if the message is

[3] GLUE uses `www.themindelectric.com` in quite a few of the namespace definitions it generates in the WSDL. You can override the `targetNamespace` by calling `context.addProperty( )` with "targetNamespace" as the property name, and the value you want as the second parameter. However, that won't eliminate the use of `www.themindelectric.com` in other sections of the WSDL.

an out parameter. From the standpoint of the message element, there is no such thing as an in/out parameter; all messages are sent either into a service, or out from it.

Each message contains 0 or more part elements. For input messages of an RPC-style service like ours, each part represents a method parameter. The service's out parameters won't appear in the description of an input message, since no associated value would be passed from the client to the server. In this example, the message named getQuote0SoapIn contains a single part named arg0 of type string. That message part corresponds to the first (and only) parameter of the getQuote service method: a string containing the stock symbol. The getQuoteSoapOut message also contains a single part, this time named Result. That's the return value of the getQuote method. Notice that Result is defined as an instance of type Quote, and that it is namespace-qualified using http://www.themindelectric.com/package/javasoap.book.ch9.services/. This refers to the custom quote type defined earlier in the WSDL document. The getHeadlines messages follow the same pattern, so we won't discuss them. If there were any out or in/out parameters that returned values in any of these messages, they would be defined as part elements immediately following the Result part of the corresponding xxxSoapOut message. For instance, if the getQuote method used an in/out parameter for the stock symbol, the message element for the outbound response would look like this:

```
<message name="getQuote0SoapOut">
 <part name="Result"
    xmlns:ns1="http://www.themindelectric.com/package/
                        javasoap.book.ch9.services/"
    type="ns1:Quote" />
 <part name="arg0" type="xsd:string" />
</message>
```

The portType element is an abstraction that defines a set of operations and their associated messages. Our portType has a name attribute with a value "CorpDataServiceSoap". In this file, the name is derived by appending "Soap" to the name used earlier in the WSDL document. There are two operation elements inside portType, each corresponding to a service method. The first one contains a name attribute with a value of "getQuote", which is the service name. It also contains a parameterOrder attribute that specifies the order in which the method parameters should appear. The getQuote method has only one parameter, so that value is "arg0". For operations that contain more than one parameter, the value of the parameterOrder attribute lists each parameter, separated by spaces. Each operation contains a single input and a single output element, each of which contains a name and a message attribute. The name contains the name of the message, and the message contains the namespace-qualified name of the message element defined earlier in the WSDL.

The next major section of the WSDL document is binding. This section combines the abstract portType with a physical protocol. The name attribute of the binding is, once again, CorpDataServiceSoap. The type attribute refers to the port type being implemented by this binding; in this case it refers back to the portType element defined in the previous WSDL section. Under the binding element we find another binding tag, but this one is namespace-qualified by the soap namespace identifier defined at the root of the document. This element specifies the SOAP binding information. The style attribute has a value of "rpc", specifying that this is an RPC-style service binding. The transport attribute specifies the underlying transport; in this case, the transport is http://schemas.xmlsoap.org/soap/http, which represents the HTTP transport. Following the soap:binding element are operation elements for each of the service operations, or methods, present in the binding. Let's look at the first one, which has a name attribute with the value "getQuote". It contains an element called soap:operation, which has two attributes. The style attribute has the value "rpc", which means that this binding represents an RPC-style SOAP invocation. The soapAction attribute has the value "getQuote", which is the name of the method itself. This is a subtle but important part of SOAP interoperability. It's necessary to send the correct value for the SOAPAction header field when invoking services; this is an area where you could run into trouble with some SOAP implementations if you aren't paying attention. In this example, the attribute tells us that the service wants the name of the method being invoked to be supplied in the SOAPAction header field. If you don't supply the appropriate header, things aren't going to work. Each operation also has an input and an ouput element. These two elements share the same structure, so we'll only look at the input. The input element has a name attribute with the value "getQuote0SoapIn". Recognize that name? That's the message that the WSDL defined earlier; now we're associating the message with this binding operation. There's also a soap:body element that specifies the namespace of the binding and the type of encoding used. You can see that this example uses standard SOAP encoding, and that the namespace is the service name itself; urn:CorpDataServices. We set this namespace in the CorpDataServices class by calling the context.setProperty( ) method.

The last stop in our WSDL example is the service element. This element combines a binding with a specific address. The name attribute contains the service name, which in this case is CorpDataService. This is followed by a documentation tag that contains a brief textual description of the service. The data contained under this element came from the context.addProperty("description", "Corporate Services Demo") method call we made in the CorpDataServices application that published this service. If you don't provide a service description, GLUE derives a description based on the name of the implementing Java class. The last element is named port , and it's the one that combines the binding with the physical address. The name attribute is set to "CorpDataServiceSoap". Although this is just a name and has no real implications, at this stage of the WSDL I like this style of naming because it tells me immediately that I'm talking about the CorpDataService using a SOAP binding. Anyway, the binding attribute is set to tns:CorpDataServiceSoap, which refers to the binding that we reviewed earlier. The soap:address element contains a location attribute that contains the actual location of the service. For our service, the location is *http://mindstrm.com:8004/glue/urn:CorpDataServices*, the URL where the service has been published.

9.2 Calling a GLUE Service from an ApacheSOAP Client

Since we already have a GLUE service running, let's try to access it with a client written using Apache SOAP. It's nice of the service to publish WSDL, but Apache SOAP clients can't do anything with that data. So we'll have to derive whatever information we need from the WSDL ourselves. Before we get into that, though, let's write a quick application that tries to call the getHeadlines method of the urn:CorpDataServices service.

First we need to determine the proper URL. When we were calling Apache SOAP services, the URL pointed to the rpcrouter servlet. In this case, a GLUE server application is hosting the service, so the URL we want is *http://mindstrm.com:8004/glue/urn:CorpDataServices*.

With pure Apache SOAP examples, we always called `call.setTargetObjectURI( )` and passed the service name as the parameter, but here we've already specified the service name in the URL. It turns out that this is the right choice, but we've got to call `setTargetObjectURI( )` with some value or we'll get a client-side error. GLUE doesn't appear to care what value we use, since we've already arrived at the target endpoint, so we can just use any nonempty string when we call `setTargetObjectURI( )`. I'm inclined to use the service name again, but it seems to work no matter what we do. Here's the code for the `Apache2Glue` application:

```
package javasoap.book.ch9;
import java.net.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import org.apache.soap.encoding.*;
import org.apache.soap.util.xml.*;
import org.w3c.dom.*;
public class Apache2Glue {
   public static void main(String[] args)
      throws Exception {
      URL url =
         new URL("http://mindstrm.com:8004/glue/urn:CorpDataServices");

      Call call = new Call( );
      call.setTargetObjectURI("XYZ");

      call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
      String stock = args[0];
      Vector params = new Vector( );
      params.addElement(new Parameter("stock", String.class,
                                stock, null));
      call.setParams(params);

      try {
         call.setMethodName("getHeadlines");
         Response resp = call.invoke(url, "");
         if (resp.generatedFault( )) {
            Fault fault = resp.getFault( );
            String code = fault.getFaultCode( );
            String desc = fault.getFaultString( );
            System.out.println(code + ": " + desc);
            Vector v = fault.getDetailEntries( );
            int cnt = v.size( );
            for (int i = 0; i < cnt; i++) {
               Element n = (Element)v.elementAt(i);
               Node nd = n.getFirstChild( );
               System.out.println(n.getNodeName( ) + ": " +
                                   nd.getNodeValue( ));
            }
         }
         else {
            Parameter ret = resp.getReturnValue( );
            String[] value = (String[])ret.getValue( );
            int cnt = value.length;
            for (int i = 0; i < cnt; i++) {
               System.out.println(value[i]);
            }
         }
      }
      catch (SOAPException e) {
         System.err.println("Caught SOAPException (" +
                        e.getFaultCode(  ) + "): " +
                        e.getMessage(  ));
      }
   }
}
```

The rest of the code is the same as if we were accessing an Apache SOAP server. That wasn't very painful. It's just a matter of understanding the location of the service. If you look back at the WSDL for this service, you'll find that the answer is right in front of you: the `location` attribute of the `soap:address` element in the `service` section tells you the URL. The fact that GLUE doesn't really care about the target object URI is interesting, but it certainly won't get you into any trouble.

If you run the `Apache2Glue` application with a valid stock symbol as the command- line parameter, it spits back some headlines of news stories related to the stock symbol. For example, the following command asks for headlines related to General Motors:

```
java javasoap.book.ch9.clients.Apache2Glue GM
```

Here's the result:

```
GMAC files shelf for $8.1 billion in notes
Vivendi Pays $1.5B for EchoStar Slice
Product Is King—Again
GM plans to form venture with SAIC-Wuling in China
Vivendi Pays $1.5B for EchoStar Slice
Daewoo Mtr makes first loan payment in a year-report
'Twas the Night Before Q4
Valeo Says U.S. Unit Filed for Bankruptcy
Honda's tank in the SUV wars
Daewoo Says to Resume Operations Monday
```

Since my service takes these headlines from a live feed, you'll get different results when you run this program.

Now that we have a working client, let's modify it to call the `getQuote()` service method. In this case, we have to deal with a custom data type. So before we go any further, we must create a quote class that we can map to the value returned by the service method. The WSDL describes the custom quote type for us. Here's the relevant part of the document:

```
<complexType name="Quote">
  <sequence>
    <element name="_symbol" nillable="true" type="string" />
    <element name="_lastPrice" type="float" />
    <element name="_change" type="float" />
    <element name="_timeStamp" nillable="true" type="string" />
    <element name="_volume" type="long" />
  </sequence>
</complexType>
```

There's no need to write a custom serializer to handle this; we can write a Java bean with appropriate set and get methods. The property names and data types must correspond to the `name` and `type` attribute values of the `elements` of the `Quote` type. For example, the class must have a property named `_symbol`, so we'll need a `get_symbol( )` method and a `set_symbol( )` method. We'll also add a `toString( )` method to make it easy to display the object's contents after we retrieve it from the service. Here's the code for the client-side `Quote` class:

```
package javasoap.book.ch9.clients;
import java.io.*;
import java.util.*;
import java.net.*;
public class Quote {
   String _symbol;
   float  _lastPrice;
   float  _change;
   String _timeStamp;
   long   _volume;
   public String get_symbol(  ) {
      return _symbol;
   }
   public void set_symbol(String symbol) {
      _symbol = symbol;
   }
   public float get_lastPrice(  ) {
      return _lastPrice;
   }
   public void set_lastPrice(float lastPrice) {
      _lastPrice = lastPrice;
   }
   public float get_change(  ) {
      return _change;
   }
   public void set_change(float change) {
      _change = change;
   }
   public String get_timeStamp(  ) {
      return _timeStamp;
   }
   public void set_timeStamp(String timeStamp) {
      _timeStamp = timeStamp;
   }
   public long get_volume(  ) {
      return _volume;
   }
   public void set_volume(long volume) {
      _volume = volume;
   }
   public String toString(  ) {
      String result = "Symbol:     " + get_symbol(  ) +
                      "\nLast Price: " + get_lastPrice(  ) +
                      "\nChange:     " + get_change(  ) +
                      "\nTime Stamp: " + get_timeStamp(  ) +
                      "\nVolume:     " + get_volume(  );
      return result;
   }
}
```

Now we're ready to modify the client program. I'll call the new class `Apache2Glue_v2`. We need to map the data returned from `getQuote` to the `Quote` class. To make this mapping, we need to know the namespace that the data type comes from. Once again, the WSDL gives us this information:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:tns=
    "http://www.themindelectric.com/package/javasoap.book.ch9.services/"
  targetNamespace=
    "http://www.themindelectric.com/package/javasoap.book.ch9.services/">
  <complexType name="Quote">
    <sequence>
      <element name="_symbol" nillable="true" type="string" />
      <element name="_lastPrice" type="float" />
      <element name="_change" type="float" />
      <element name="_timeStamp" nillable="true" type="string" />
      <element name="_volume" type="long" />
    </sequence>
  </complexType>
</schema>
```

The `targetNamespace` for the custom quote type is defined in the WSDL document as `http://www.themindelectric.com/package/javasoap.book.ch9.services/`. We use this namespace in the `QName` constructor when we map the data to our quote class, but that's getting somewhat ahead of the story. The client creates an instance of `SOAPMappingRegistry` and an instance of `BeanSerializer`, just as in Chapter 5. We then use the mapping registry's `mapTypes( )` method to map the object returned by `getQuote` to our `Quote` object. Next, we set up the method call in the standard way, make the call, and pass the `Quote` object we get back to `System.out.println( )`. Here's the code for `Apache2Glue_v2`:

```
package javasoap.book.ch9.clients;
import java.net.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import org.apache.soap.encoding.*;
import org.apache.soap.encoding.soapenc.*;
import org.apache.soap.util.xml.*;
import org.w3c.dom.*;
public class Apache2Glue_v2 {
   public static void main(String[] args)
      throws Exception {

      URL url =
          new URL("http://mindstrm.com:8004/glue/urn:CorpDataServices");
      SOAPMappingRegistry smr = new SOAPMappingRegistry(  );
      BeanSerializer beanSer = new BeanSerializer(  );
      smr.mapTypes(Constants.NS_URI_SOAP_ENC,
        new QName(
          "http://www.themindelectric.com/package/" +
          "javasoap.book.ch9.services/",
          "Quote"),
          Quote.class, beanSer, beanSer);

      Call call = new Call(  );
      call.setSOAPMappingRegistry(smr);
      call.setTargetObjectURI("XYZ");
      call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
      String stock = args[0];
      Vector params = new Vector(  );
      params.addElement(new Parameter("stock", String.class,
                                  stock, null));
      call.setParams(params);
      try {
         call.setMethodName("getQuote");
         Response resp = call.invoke(url, "");
         if (resp.generatedFault( )) {
            Fault fault = resp.getFault( );
            String code = fault.getFaultCode( );
            String desc = fault.getFaultString( );
            System.out.println(code + ": " + desc);
            Vector v = fault.getDetailEntries( );
            int cnt = v.size( );
            for (int i = 0; i < cnt; i++) {
               Element n = (Element)v.elementAt(i);
               Node nd = n.getFirstChild( );
               System.out.println(n.getNodeName( ) + ": "
                                            + nd.getNodeValue( ));
            }
         }
         else {
            Parameter ret = resp.getReturnValue( );
            Quote value = (Quote)ret.getValue( );
            System.out.println(value);
         }
      }
      catch (SOAPException e) {
         System.err.println("Caught SOAPException (" +
                           e.getFaultCode( ) + "): " +
                           e.getMessage( ));
      }
   }
}
```

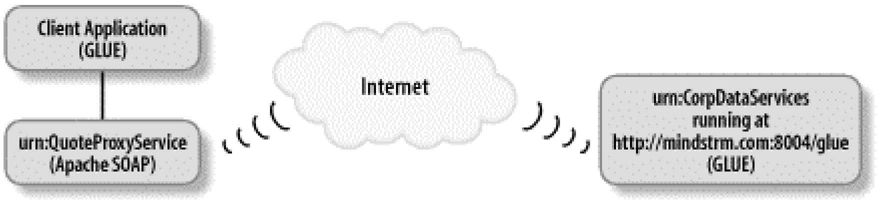
Here's the result:

```
Symbol:     GM
Last Price: 47.28
Change:     -0.2
Time Stamp: 4:03PM
Volume:     3490100
```

9.3 A Proxy Service Using Apache SOAP

I'm going to use *http://mindstrm.com:8004/glue/urn:CorpDataServices* as the backend for the services we develop through the rest of this chapter. In this example we'll create an Apache SOAP service called `urn:QuoteProxyService` that gets its data from the GLUE-based service we've been working with. In other words, this new Apache SOAP service acts as a proxy to the quote retrieval part of the GLUE-based service. This setup is depicted in Figure 9-2.

Figure 9-2. A proxy service


Client Application (GLUE)
urn:QuoteProxyService (Apache SOAP)
Internet
urn:CorpDataServices running at http://mindstrm.com:8004/glue (GLUE)

Instead of simply replicating the `getQuote()` service method from `urn:CorpDataServices`, let's do something more interesting. After all, we're free to design the proxy service any way we want; the fact that it gets its data from another service is really just an implementation detail. So let's add some value to `urn:QuoteProxyService` by giving it a method that can handle multiple stock symbols in a single call. This facility would be convenient for client applications that work with multiple stocks at one time. The method name is `getQuotes()`; it takes a single string array as its parameter. Each element of the array is a stock symbol, and the return value of the method is an array of quotes. The new service doesn't proxy the `getHeadlines()` method of the `urn:CorpDataServices` service. There's no real reason for this; I just chose to design my proxy that way.

Once again we're going to need a Java quote class for the proxy server to return. To be consistent with our package naming convention, this class belongs in `javasoap.book.ch9.services`. To avoid any confusion with earlier Java quote classes, let's name this one `ProxyQuote`. This class contains the same information as the other quote classes, but the property names and accessor methods are different. The only reason for this difference is to avoid confusion; the names really don't matter. Here's the code for the `ProxyQuote` class:

```
package javasoap.book.ch9.services;
import java.io.*;
import java.util.*;
import java.net.*;
public class ProxyQuote {
   String _sym;
   float  _last;
   float  _diff;
   String _time;
   long   _vol;
   public String getStockSymbol(  ) {
      return _sym;
   }
   public void setStockSymbol(String symbol) {
      _sym = symbol;
   }
   public float getLast(  ) {
      return _last;
   }
   public void setLast(float last) {
      _last = last;
   }
   public float getDiff(  ) {
      return _diff;
   }
   public void setDiff(float diff) {
      _diff = diff;
   }
   public String getTime( ) {
      return _time;
   }
   public void setTime(String time) {
      _time = time;
   }
   public long getVol( ) {
      return _vol;
   }
   public void setVol(long vol) {
      _vol = vol;
   }
}
```

So now we have a Java class that represents the quote data from the `urn:QuoteProxyService`. But let's not forget that this service is also a client to the `urn:CorpDataService`. When we wrote an Apache SOAP client for that service, we needed a Java class to map the return quote data. We can steal this from `Quote`: just change the package name to `javasoap.book.ch9.services`, delete the `toString( )` method, and change its name to `RemoteQuote`. Here's the result:

```
package javasoap.book.ch9.services;
import java.io.*;
import java.util.*;
public class RemoteQuote {
   String _symbol;
   float  _lastPrice;
   float  _change;
   String _timeStamp;
   long   _volume;
   public String get_symbol( ) {
      return _symbol;
   }
   public void set_symbol(String symbol) {
      _symbol = symbol;
   }
   public float get_lastPrice( ) {
      return _lastPrice;
   }
   public void set_lastPrice(float lastPrice) {
      _lastPrice = lastPrice;
   }
   public float get_change( ) {
      return _change;
   }
   public void set_change(float change) {
      _change = change;
   }
   public String get_timeStamp( ) {
      return _timeStamp;
   }
   public void set_timeStamp(String timeStamp) {
      _timeStamp = timeStamp;
   }
   public long get_volume( ) {
      return _volume;
   }
   public void set_volume(long volume) {
      _volume = volume;
   }
}
```

Now we've got all the support classes we need, and can move ahead with the code for the `urn:QuoteProxyService`. The Java class is named `QuoteProxyService`, and the code is pretty simple. The service exposes one method, called `getQuotes( )`, which retrieves an arbitrary number of quotes based on the array of stock symbols. The method returns a `ProxyQuote[]`, which is an array of `ProxyQuote` instances, one for each stock symbol. Each quote is obtained by calling the `getStockQuote( )` method, which returns a single `ProxyQuote` based on the stock symbol passed as its parameter. Each quote is stored in a local `Vector`, the contents of which are used to populate an appropriately sized `ProxyQuote[]`. That array is then returned. The process used by the `getStockQuote( )` method is essentially the same one used by the `Apache2Glue_v2` client application. The only differences are that we don't bother with any returned fault elements, and we need to pull the data out of the resulting `RemoteQuote` instance in order to populate the `ProxyQuote` instance; remember that the quote data type returned by the back-end service is not the same as the quote data type that this new service returns to its clients. And that's how it should be. While it would be simplest for this example to return `RemoteQuote` instances, that's not how the service would work in the real world; a proxy service shouldn't expose the existence of the back-end service in any way. Here's the code for `QuoteProxyService`:

```
package javasoap.book.ch9.services;
import java.net.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import org.apache.soap.encoding.*;
import org.apache.soap.encoding.soapenc.*;
import org.apache.soap.util.xml.*;
import org.w3c.dom.*;
public class QuoteProxyService {
   public QuoteProxyService( ) {
   }
   public ProxyQuote[] getQuotes(String[] symbols)
                      throws Exception {
      Vector v = new Vector( );
      int cnt = symbols.length;
      for (int i = 0; i < cnt; i++) {
         v.add(getStockQuote(symbols[i]));
      }
      cnt = v.size( );
      ProxyQuote[] quotes = new ProxyQuote[cnt];
      for (int i = 0; i < cnt; i++) {
         quotes[i] = (ProxyQuote)v.elementAt(i);
      }
      return quotes;
   }
   ProxyQuote getStockQuote(String symbol)
                    throws Exception {
      URL url = new
        URL("http://mindstrm.com:8004/glue/urn:CorpDataServices");
      SOAPMappingRegistry smr = new SOAPMappingRegistry( );
      BeanSerializer beanSer = new BeanSerializer( );
      smr.mapTypes(Constants.NS_URI_SOAP_ENC,
        new QName(
          "http://www.themindelectric.com/package/" +
          "javasoap.book.ch9.services/",
          "Quote"),
          RemoteQuote.class, beanSer, beanSer);

      Call call = new Call( );
      call.setSOAPMappingRegistry(smr);
      call.setTargetObjectURI("XYZ");
      call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
      String stock = symbol;
      Vector params = new Vector( );
      params.addElement(new Parameter("stock", String.class,
                                   stock, null));
      call.setParams(params);

      call.setMethodName("getQuote");
      Response resp = call.invoke(url, "");
      ProxyQuote quote = new ProxyQuote( );
      if (resp.generatedFault( )) {
         throw new Exception("Service Call Failed");
      }
      else {
         Parameter ret = resp.getReturnValue( );
         RemoteQuote value = (RemoteQuote)ret.getValue( );
         quote.setStockSymbol(value.get_symbol( ));
         quote.setLast(value.get_lastPrice( ));
         quote.setDiff(value.get_change( ));
         quote.setTime(value.get_timeStamp( ));
         quote.setVol(value.get_volume( ));
      }
      return quote;
   }
}
```

We can deploy this service the same way we've deployed all the Apache SOAP services we've created throughout the book. We'll need to include a mapping for the `ProxyQuote` class. The fully qualified name for the quote type is `urn:QuoteProxyService:Quote`. We used a JavaBeans-compliant property naming scheme, so the mapping uses the `BeanSerializer` class for the conversions. Here's what the deployment descriptor looks like:

```
<isd:service xmlns:isd="http://xml.apache.org/xml-soap/deployment"
             id="urn:QuoteProxyService">
  <isd:provider type="java"
                scope="Application"
                methods="getQuotes">
    <isd:java class="javasoap.book.ch9.services.QuoteProxyService"
    static="false"/>
  </isd:provider>
  <isd:faultListener>org.apache.soap.server.DOMFaultListener
  </isd:faultListener>
  <isd:mappings>
    <isd:map encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     xmlns:x="urn:QuoteProxyService" qname="x:Quote"
     javaType="javasoap.book.ch9.services.ProxyQuote"
     java2XMLClassName="org.apache.soap.encoding.soapenc.BeanSerializer"
     xml2JavaClassName="org.apache.soap.encoding.soapenc.BeanSerializer"/>
  </isd:mappings>
</isd:service>
```

9.4 Calling an Apache SOAP Service from a GLUE Client

This time, we'll write a GLUE client that accesses the `urn:QuoteProxyService` we just created. But how do we begin? GLUE creates service bindings dynamically based on a WSDL document. In our GLUE-to-GLUE examples, GLUE automatically generated the WSDL document describing the service, but Apache SOAP doesn't do that. We could certainly write an appropriate WSDL document by hand . . . but that's a lot of work, and the chances of ending up with a WSDL document that's correct are fairly small, unless you want to learn a lot more about WSDL than anyone should have to know. Instead, we'll use the `java2wsdl` tool that comes with GLUE to generate the WSDL.[4] This tool doesn't care that the Java code isn't for a GLUE-based service; it simply reads the Java code and generates the WSDL. `java2wsdl` doesn't give us a perfect WSDL document for the Apache-based service, but it's much less work to modify the document by hand than it is to write the whole thing. So let's generate the WSDL:

```
java2wsdl javasoap.book.ch9.services.QuoteProxyService -n
    urn:QuoteProxyService -t urn:QuoteProxyService -s
    -e http://georgetown:8080/soap/servlet/rpcrouter
```

The first parameter to `java2wsdl` is the Java class that implements the service. The `-n` option allows us to specify the namespace; we'll use the name of the service as the namespace. The `-t` option allows us to specify the value of the `targetNamespace` attribute. These two options are applied only to the `definitions` element of the generated WSDL. We'll need to modify the rest by hand, as you'll see shortly. The `-s` option tells the tool to generate a SOAP binding. We need this option because we are creating a service definition for a known endpoint (the location where the service was deployed), and therefore we know what transport we're using. (It's reasonably common to define the interface to the service and its binding separately, so that the same service can be deployed with different endpoints and using different protocols.) For this reason, we also include the `-e` option, which allows us to specify that endpoint. Since our RPC-style Apache SOAP services are all routed through the rpcrouter servlet, the endpoint is *http://georgetown:8080/soap/servlet/rpcrouter*.[5]

Now that we've generated a WSDL file, let's make the needed adjustments. The modified parts of the file are shown in bold.

```
<?xml version='1.0' encoding='UTF-8'?>
<definitions
name='urn:QuoteProxyService'
targetNamespace='urn:QuoteProxyService'
xmlns:tns='urn:QuoteProxyService'
xmlns:soap='http://schemas.xmlsoap.org/wsdl/soap/'
xmlns:http='http://schemas.xmlsoap.org/wsdl/http/'
xmlns:mime='http://schemas.xmlsoap.org/wsdl/mime/'
xmlns:xsd='http://www.w3.org/2001/XMLSchema'
xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/'
xmlns:wsdl='http://schemas.xmlsoap.org/wsdl/'
xmlns='http://schemas.xmlsoap.org/wsdl/'>
  <types>
    <schema xmlns='http://www.w3.org/2001/XMLSchema'
       xmlns:tns='urn:QuoteProxyService'
       targetNamespace='urn:QuoteProxyService'>
```

[4] Other tools for this purpose are available from other sources. For example, the IBM web services toolkit includes a similar utility.

[5] This location is part of my local network. You'll need to use your own appropriate network address for the location you used to deploy the service.

```
    <complexType name='Quote'>
      <sequence>
        <element name='stockSymbol' nillable='true' type='string'/>
        <element name='last' type='float'/>
        <element name='diff' type='float'/>
        <element name='time' nillable='true' type='string'/>
        <element name='vol' type='long'/>
      </sequence>
    </complexType>
    <complexType name='ArrayOfQuote'>
      <complexContent>
        <restriction base='soapenc:Array'>
          <attribute ref='soapenc:arrayType'
               wsdl:arrayType='tns:Quote[]'/>
        </restriction>
      </complexContent>
    </complexType>
  </schema>
  <schema xmlns='http://www.w3.org/2001/XMLSchema'
      xmlns:tns='urn:QuoteProxyService'
      targetNamespace='urn:QuoteProxyService'>
    <complexType name='ArrayOfstring'>
      <complexContent>
        <restriction base='soapenc:Array'>
          <attribute ref='soapenc:arrayType' wsdl:arrayType='string[]'/>
        </restriction>
      </complexContent>
    </complexType>
  </schema>
</types>
<message name='getQuotes0SoapIn'>
  <part name='arg0'
     xmlns:ns1='urn:QuoteProxyService' type='ns1:ArrayOfstring'/>
</message>
<message name='getQuotes0SoapOut'>
  <part name='Result'
     xmlns:ns1='urn:QuoteProxyService' type='ns1:ArrayOfQuote'/>
</message>
<portType name='QuoteProxyServiceSoap'>
  <operation name='getQuotes' parameterOrder='arg0'>
    <input name='getQuotes0SoapIn' message='tns:getQuotes0SoapIn'/>
    <output name='getQuotes0SoapOut' message='tns:getQuotes0SoapOut'/>
  </operation>
</portType>
<binding name='QuoteProxyServiceSoap' type='tns:QuoteProxyServiceSoap'>
  <soap:binding style='rpc'
      transport='http://schemas.xmlsoap.org/soap/http'/>
  <operation name='getQuotes'>
    <soap:operation soapAction='getQuotes' style='rpc'/>
    <input name='getQuotes0SoapIn'>
      <soap:body use='encoded' namespace='urn:QuoteProxyService'
           encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'/>
    </input>
    <output name='getQuotes0SoapOut'>
      <soap:body use='encoded' namespace='urn:QuoteProxyService'
           encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'/>
    </output>
  </operation>
</binding>
```

```
  <service name='QuoteProxyService'>
    <port name='QuoteProxyServiceSoap'
           binding='tns:QuoteProxyServiceSoap'>
      <soap:address
           location='http://georgetown:8080/soap/servlet/rpcrouter'/>
    </port>
  </service>
</definitions>
```

We need to modify the namespace and target namespace values for all of the elements except `definitions`; this element was handled correctly because of the `-n` and `-t` command-line parameters we supplied to `java2wsdl`. That's why you see modified values for all of the `namespace` and `targetNamespace` attributes. The values generated contain names based on the Java classes used by the service implementation. They're just names, but I don't want that kind of information exposed. In other parts of the document, I stripped similar Java class package names that preceded otherwise acceptable names. For example, the value of the `name` attribute of the `binding` element was preceded by the name of the package where our service classes reside. The `name` modifications weren't necessary for interoperability, but I think they eliminate any confusion regarding the need to expose Java class and package names.

A few modifications are needed in the definition of the custom quote type. The generated name for the type was `ProxyQuote`, which comes from the Java class name that implements the custom type on the server side. However, that's not the name we chose for the type in the mapping section of the deployment descriptor. Let's take another look at the relevant part of the deployment descriptor:

```
<isd:map encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:x="urn:QuoteProxyService" qname="x:Quote"
javaType="javasoap.book.ch9.services.ProxyQuote"
java2XMLClassName="org.apache.soap.encoding.soapenc.BeanSerializer"
xml2JavaClassName="org.apache.soap.encoding.soapenc.BeanSerializer"/>
```

The namespace is `urn:QuoteProxyService`. We already modified the corresponding part of the WSDL, so that part is good. But look at the `qname` attribute. We gave it a value of `Quote`,[6] not `ProxyQuote`. That means that when the response envelope is encoded, the quote data will be typed as a `Quote`, namespace-qualified by `urn:QuoteProxyService`. The `java2wsdl` generator has no way of knowing about that because the information resides outside of the Java code itself.

Finally, we need to change the names of the data elements that make up the custom quote type. The `java2wsdl` utility doesn't use JavaBeans property accessors by default; instead it looks directly at the member variables in the implementing class. So the names it used were `sym`, `_last`, etc. But the `BeanSerializer` class that encodes instances of the `ProxyQuote` class uses property names based on the JavaBeans naming standards. Well, almost. They actually aren't encoded as expected. If you were to look at the SOAP envelope returned by the service, you'd see that the elements of the custom quote data type all use lowercase for the first letter of the property name. According to the Java Beans specification, read and write accessors named `setThing( )` and `getThing( )` mean that the name of the property is `Thing`. But the `BeanSerializer` encodes them as `thing`. This problem hasn't affected us with Apache-to-Apache examples, but it does make a difference to GLUE clients. So we have

[6] The value is actually `x:Quote`, but for the sake of clarity I'm referring only to the local part of the name instead of the fully qualified name.

some minor problems introduced by quirks of both SOAP implementations. To correct these problems, the element names are changed to be `stockSymbol`, `last`, `diff`, `time`, and `vol`.

There's also a definition for the array of quotes, which is what the `getQuotes` service method returns. The value of the `name` attribute is changed to `ArrayOfQuote`, and the value of `wsdl:arrayType` is changed to `tns:Quote[]`. Essentially we're removing "Proxy" from the definition of the quote and associated quote array. The second `message` definition refers to the quote array, so it needs to be modified as well to reflect the proper name of the quote array type.

Now we need to publish this WSDL document. My Apache SOAP engine is hosted by an Apache Tomcat server. This server is listening on port 8080 and running on a machine named `georgetown` on my local network. The *webapps/soap* directory under my Apache SOAP installation directory is a perfectly good place to put the WSDL file. This means I can access it using the URL *http://georgetown:8080/soap/QuoteProxyService.wsdl*.

At this point, we're ready to write a GLUE client to access the `urn:QuoteProxyService`. All the classes for this application should be in the `javasoap.book.ch9.clients` package. We're not going to use the `wsdl2java` utility to create the classes and interfaces for binding to this service because we can't control the class names. We need the ability to control the class names because we're putting all of our client-side classes in the `javasoap.book.ch9.clients` package, and we already have a `Quote` class; we want the local Java class that gets mapped to the quote data type to be named `ProxyQuote` instead. You could run the `wsdl2java` from a directory other than the one corresponding to the `javasoap.book.ch9.clients` package; rename the generated `Quote` class and filename to `ProxyQuote`; change the references to it in the *QuoteProxyService.map* file; then move the files into the package directory. Here, though, we'll just write the classes by hand. It's not difficult, and it'll give us another chance to look inside the pieces needed for the GLUE client application.

So, let's create a client-side class named `ProxyQuote` to represent the quote data that's returned by the service. This class needs a public data field for each element of the quote, and we'll add a `toString( )` method to make it easy to display. Here's the code for the `ProxyQuote` class:

```
package javasoap.book.ch9.clients;
public class ProxyQuote {
   public String stockSymbol;
   public float last;
   public float diff;
   public String time;
   public long vol;
   public String toString( ) {
      String result = "Symbol:      " + stockSymbol +
                      "\nLast Price: " + last +
                      "\nChange:     " + diff +
                      "\nTime Stamp: " + time +
                      "\nVolume:     " + vol;
      return result;
   }
}
```

We'll need a map file that describes the mapping from the data returned by the service to the `ProxyQuote` class. The file is named *QuoteProxyService.map*. I put it in the same directory as the `javasoap.book.ch9.clients` classes so that it will be loaded automatically. Here's the map file:

```
<?xml version='1.0' encoding='UTF-8'?>
<mappings xmlns='http://www.themindelectric.com/schema/'>
  <schema xmlns='http://www.w3.org/2001/XMLSchema'
      targetNamespace='urn:QuoteProxyService'
      xmlns:electric='http://www.themindelectric.com/schema/'>
    <complexType
          name='Quote'
          electric:class='javasoap.book.ch9.clients.ProxyQuote'>
      <sequence>
        <element name='stockSymbol' nillable='true'
                 electric:field='stockSymbol' type='string'/>
        <element name='vol'
                 electric:field='vol' type='long'/>
        <element name='last'
                 electric:field='last' type='float'/>
        <element name='diff'
                 electric:field='diff' type='float'/>
        <element name='time' nillable='true'
                 electric:field='time' type='string'/>
      </sequence>
    </complexType>
  </schema>
</mappings>
```

Note that we've given the `name` attribute of the `complexType` element a value of `Quote`. That matches what we used in the deployment descriptor for the Apache service, and it matches the modification we made to the WSDL file. The `name` attributes of all the `element` tags match the names of the elements of the quote data coming from the service, and the `electric:field` attribute values match the member variable names of the mapped `ProxyQuote` class. The `targetNamespace` attribute that governs this entire mapping document has a value of `urn:QuoteProxyService`. That's the value expected to be used by the service for namespace-qualifying the return `Quote` type; and that matches the WSDL as well.

Here's a Java interface that we'll use to bind to the service. It contains the one and only service method, `getQuotes( )`.

```
package javasoap.book.ch9.clients;
public interface IQuoteProxyService {
  ProxyQuote[] getQuotes(String[] symbols);
}
```

We now have everything we need to write a client application. We'll use the command-line arguments to specify the stock symbols for which we want quotes. We'll bind to the service using the address of the WSDL document we published, invoke the `getQuotes( )` method using `args` as the parameter, and display the results by iterating over the returned array. Here's the code:

```
package javasoap.book.ch9.clients;
import electric.registry.Registry;
```

```
public class Glue2Apache {
   public static void main( String[] args )
            throws Exception {
       IQuoteProxyService service = (IQuoteProxyService)Registry.bind(
              "http://georgetown:8080/soap/QuoteProxyService.wsdl",
              IQuoteProxyService.class);
       ProxyQuote[] quotes = service.getQuotes(args);
       int cnt = quotes.length;
       for (int i = 0; i < cnt; i++) {
          System.out.println(quotes[i]);
          System.out.println("");
       }
   }
}
```

I ran the service with four stock symbols on the command line:

```
java javasoap.book.ch9.clients.Glue2Apache IBM MSFT GM MMM
```

Here are the results I got:

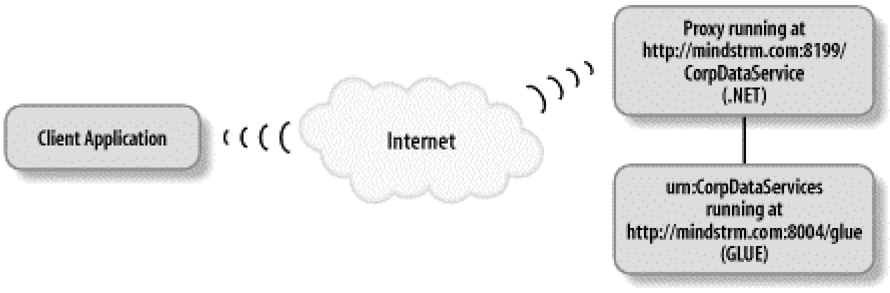
```
Symbol:     IBM
Last Price: 123.1
Change:     -0.79
Time Stamp: 1:59PM
Volume:     3916800
Symbol:     MSFT
Last Price: 67.701
Change:     -1.789
Time Stamp: 2:04PM
Volume:     21485800
Symbol:     GM
Last Price: 47.36
Change:     -0.74
Time Stamp: 1:59PM
Volume:     1719700
Symbol:     MMM
Last Price: 118.51
Change:     -1.29
Time Stamp: 2:00PM
Volume:     953400
```

9.5 Accessing .NET Services

Although it's not based on Java, there's no way to avoid a discussion of Microsoft's .NET technology. If you're interacting with web services, you'll certainly run into .NET more than once. To show how to achieve interoperability between .NET and web services written in Java, I've written a .NET web service using the C# language and published it on the Internet at http://mindstrm.com:8199/CorpDataService/Proxy.asmx.[7] For those of you working with .NET, it's necessary to specify that your service style is RPC, since that is not the .NET default service style. To do so, use the `[SoapRpcService( )]` declaration in your C# code. My .NET service acts as a proxy to the `urn:CorpDataServices` service running at *http://mindstrm.com:8004/glue/urn:CorpDataServices*. Figure 9-3 depicts the relationships involved.

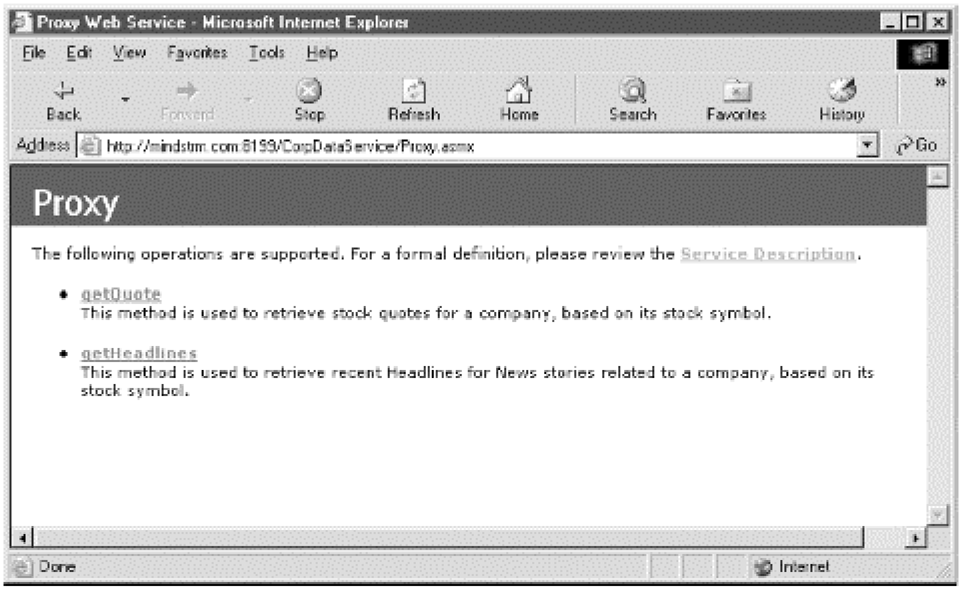
[7] The code for the service is located at http://www.mindstrm.com/soap.htm.

Figure 9-3. Interactions between a client, a proxy, and a service


Client Application
Internet
Proxy running at http://mindstrm.com:8199/CorpDataService (.NET)
urn:CorpDataServices running at http://mindstrm.com:8004/glue (GLUE)

The `Proxy` service has a `getHeadlines` method as well as a `getQuote` method, just like the `urn:CorpDataServices` service. Each method takes a single string parameter containing a stock symbol. `getHeadlines` returns an array of strings, and `getQuote` returns a custom quote data type. You can find information about .NET services simply by navigating to their URL. So enter the URL http://mindstrm.com:8199/CorpDataService/Proxy.asmx in your browser and you'll end up at the page shown in Figure 9-4.

Figure 9-4. Information about a .NET service


Proxy Web Service - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History
Address http://mindstrm.com:8199/CorpDataService/Proxy.asmx
Go
Done
Internet

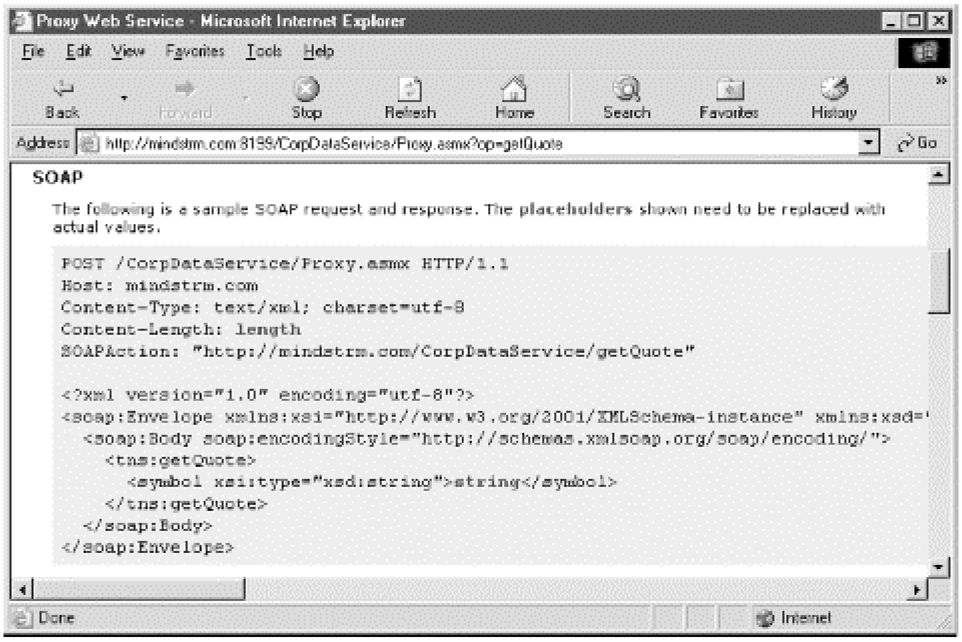
Proxy

The following operations are supported. For a formal definition, please review the Service Description.

- getQuote
  This method is used to retrieve stock quotes for a company, based on its stock symbol.
- getHeadlines
  This method is used to retrieve recent Headlines for News stories related to a company, based on its stock symbol.

If you click on the getQuote link or the getHeadlines link, you'll be taken to a page that allows you to see examples of the XML for sending requests to the service method, as well as the associated responses. There's also a basic user interface that allows you to use the service method without having to write any code. Clicking on Service Description gives you the WSDL for the entire `Proxy` service. We could look through the WSDL to find the required `SOAPAction`, namespaces, etc., so that we can set up our Apache SOAP client correctly. However, there's an easier way. Since clicking on the name of a method takes you to a page that shows the XML for messages passed between a client and the service method, we can use the browser to figure out what sorts of messages to send. Figure 9-5 shows the section of that page that shows what the request envelope should look like for invoking `getQuote`.

Figure 9-5. The SOAP document sent to getQuote


Proxy Web Service - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Forward
Stop
Refresh
Home
Search
Favorites
History
Address
http://mindstrm.com:8199/CorpDataService/Proxy.asmx?op=getQuote
Go
Done
Internet

SOAP

The following is a sample SOAP request and response. The placeholders shown need to be replaced with actual values.

```
POST /CorpDataService/Proxy.asmx HTTP/1.1
Host: mindstrm.com
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://mindstrm.com/CorpDataService/getQuote"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd='
  <soap:Body soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <tns:getQuote>
      <symbol xsi:type="xsd:string">string</symbol>
    </tns:getQuote>
  </soap:Body>
</soap:Envelope>
```

This gives us just about everything we need to access the service, except for its address. Although the `Host` field of the HTTP header shows that the server name is `mindstrm.com`, it doesn't show which port to use to access the service. To get the proper address, we need to peek at the WSDL. Here's the `service` section of the WSDL document for this service:

```
<service name="Proxy">
  <port name="ProxySoap" binding="s2:ProxySoap">
  <soap:address
    location="http://mindstrm.com:8199/CorpDataService/Proxy.asmx" />
  </port>
```

So now we see that the URL is *http://mindstrm.com:8199/CorpDataService/Proxy.asmx*. Next we need the value of the `SOAPAction` header field. Yes, in this case it matters! If you don't get this right, .NET will not invoke your service. So we need to specify the `SOAPAction` in our application, using the value `http://mindstrm.com/CorpDataService/getQuote`. Remember, this is not an address to be used as the location of a service or service method, and therefore doesn't include details like a port number. It's a URI, a name that indicates to the .NET framework what the enclosed SOAP envelope is about.

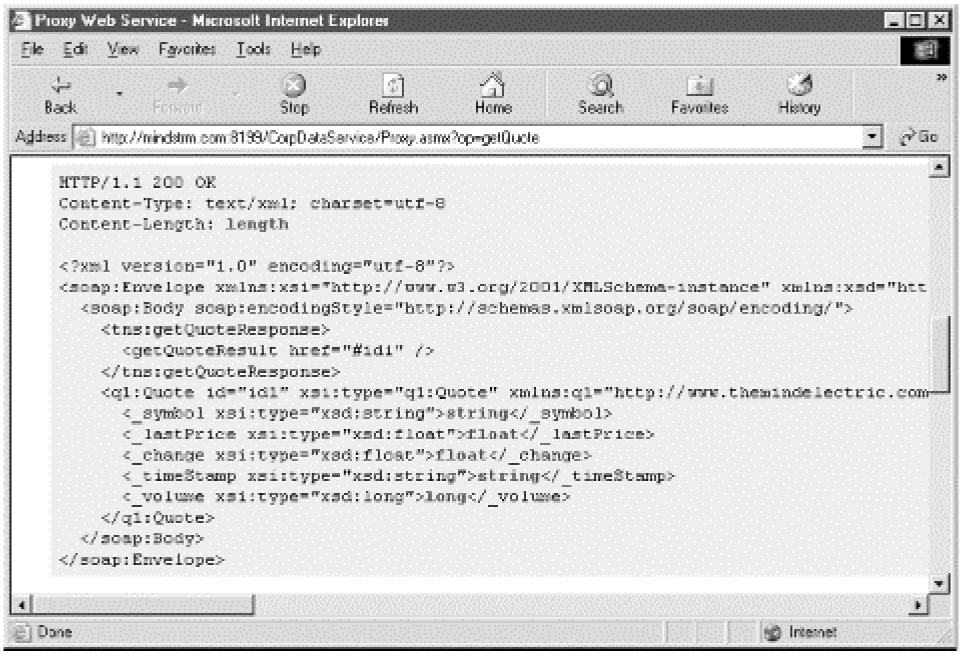
Now we need to determine the namespace used to qualify the `getQuote` element. `getQuote` is qualified using the namespace identifier `tns`. (This doesn't show up in Figure 9-5, but you'll see it if you scroll to the right.) Here's the namespace declaration:

```
xmlns:tns=http://mindstrm.com/CorpDataService
```

This declaration tells us that the namespace we want is `http://mindstrm.com/CorpDataService`. The last thing to notice is the name of the parameter being passed to the `getQuote` method. Its name is `symbol`, and it matters to .NET.

Before we write any code, we should look at the response that the service will send back. This response tells us what we need to know about the custom quote data type we have to deal with. Figure 9-6 shows an example of the response envelope.

Figure 9-6. The response to a call to getQuote


Proxy Web Service - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History
Address http://mindstrm.com:8199/CorpDataService/Proxy.asmx?op=getQuote
Go
Done
Internet

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="htt
  <soap:Body soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <tns:getQuoteResponse>
      <getQuoteResult href="#id1" />
    </tns:getQuoteResponse>
    <q1:Quote id="id1" xsi:type="q1:Quote" xmlns:q1="http://www.themindelectric.com
      <_symbol xsi:type="xsd:string">string</_symbol>
      <_lastPrice xsi:type="xsd:float">float</_lastPrice>
      <_change xsi:type="xsd:float">float</_change>
      <_timeStamp xsi:type="xsd:string">string</_timeStamp>
      <_volume xsi:type="xsd:long">long</_volume>
    </q1:Quote>
  </soap:Body>
</soap:Envelope>
```

`getQuoteResponse` is qualified by `tns`, which has the same value as it does in the request envelope: `http://mindstrm.com/CorpDataService`. (The namespace isn't on the visible part of the page.) Also note that the `Quote` element is namespace-qualified as well, using the identifier `q1`. Again, the namespace is cut off, so here's what it actually contains:

```
xmlns:q1="http://www.themindelectric.com/package/javasoap.book.ch9.services
/"
```

What's `www.themindelectric.com` doing in this XML? When developing the .NET service, I used .NET Visual Studio to generate a C# class that accesses the `urn:CorpDataServices` service implemented in GLUE. It does so by requesting the WSDL, which is generated automatically by the GLUE-based server. That server uses `www.themindelectric.com` and the Java package name when it creates the namespace. Remember, it's just a name. Knowing the name will be convenient when we map this type to a Java class.

Here's the code for an Apache SOAP client application that accesses the .NET service:

```
package javasoap.book.ch9.clients;
import java.net.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import org.apache.soap.encoding.*;
import org.apache.soap.encoding.soapenc.*;
import org.apache.soap.util.xml.*;
import org.w3c.dom.*;
```

```
public class Apache2DotNet {
   public static void main(String[] args)
      throws Exception {

      URL url =
           new URL("http://mindstrm.com:8199/CorpDataService/Proxy.asmx");
      SOAPMappingRegistry smr = new SOAPMappingRegistry(  );
      BeanSerializer beanSer = new BeanSerializer(  );
      smr.mapTypes(Constants.NS_URI_SOAP_ENC, new QName(
        "http://www.themindelectric.com/package/" +
        "javasoap.book.ch9.services/",
        "Quote"),
         Quote.class, beanSer, beanSer);

      Call call = new Call(  );
      call.setSOAPMappingRegistry(smr);
      call.setTargetObjectURI("http://mindstrm.com/CorpDataService");
      call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
      String stock = args[0];
      Vector params = new Vector(  );
      params.addElement(new Parameter("symbol",
                                   String.class, stock, null));
      call.setParams(params);

      try {
         call.setMethodName("getQuote");
         Response resp = call.invoke(url,
               "http://mindstrm.com/CorpDataService/getQuote");
         if (resp.generatedFault(  )) {
            Fault fault = resp.getFault(  );
            String code = fault.getFaultCode(  );
            String desc = fault.getFaultString(  );
            System.out.println(code + ": " + desc);
            Vector v = fault.getDetailEntries(  );
            int cnt = v.size(  );
            for (int i = 0; i < cnt; i++) {
               Element n = (Element)v.elementAt(i);
               Node nd = n.getFirstChild(  );
               System.out.println(n.getNodeName(  ) + ": " +
                                                 nd.getNodeValue(  ));
            }
         }
         else {
            Parameter ret = resp.getReturnValue(  );
            Quote value = (Quote)ret.getValue(  );
            System.out.println(value);
         }
      }
      catch (SOAPException e) {
         System.err.println("Caught SOAPException (" +
                          e.getFaultCode(  ) + "): " +
                          e.getMessage(  ));
      }
   }
}
```

The `Apache2DotNet` Java class is similar to the `Apache2Glue_v2` class we developed earlier, so let's just review the changes instead of revisiting the entire class. The changes are shown in bold so you can pick them out easily.

The URL needed to access the service is *http://mindstrm.com:8199/CorpDataService/Proxy.asmx*. Since we're going to invoke `getQuote`, we'll get back an instance of a custom quote data type. You may have noticed in Figure 9-6 that the data fields of that quote type are exactly the same as the data fields of the `Quote` type returned by `urn:CorpDataServices`. So we can reuse some code here by mapping the `Quote` class we developed when we wrote the Apache SOAP client for accessing the GLUE service. It turns out that the namespace we need to qualify the quote type is also the same: `http://www.themindelectric.com/package/javasoap.book.ch9.services`.

The next modification is the parameter passed to the `call.setTargetObjectURI( )` method. In this case we pass `http://mindstrm.com/CorpDataService`, which is the namespace used to qualify the `getQuote` element in the WSDL. We also need to change the name of the parameter specified in the `Parameter( )` constructor to `symbol`, since that's the name we found earlier when we looked at the request envelope.

The last change is that we need to specify the value of the `SOAPAction` associated with this method invocation. To set this value, we use the second parameter of the `call.invoke( )` method. In our other Apache SOAP clients, we passed an empty string as the second parameter because we had no need to specify a `SOAPAction`. However, when invoking .NET service methods, we must include the correct `SOAPAction`. Earlier, we found that `SOAPAction` needed to have the value `http://mindstrm.com/CorpDataService/getQuote`, so that's what we pass to `call.invoke( )`.

The rest of the code is unchanged. If you run the `Apache2DotNet` application, it invokes the `getQuote` method on our .NET service, which in turn invokes the `getQuote` method on the `urn:CorpDataServices` service. Here's what the output looked like when I ran the application, passing GM as the stock symbol:

```
Symbol:     GM
Last Price: 48.05
Change:     0.35
Time Stamp: 11:06AM
Volume:     1078700
```

9.6 Writing an Apache Axis Client

The Apache group is currently working on a completely new SOAP implementation known as Axis.[8] It's really too early in the evolution of that project to spend much time on it. It's likely to change quite a bit before it stabilizes, and right now it's not ready for production use. The plan is for Axis to conform to JAX-RPC, an emerging API standard for SOAP RPC. Nevertheless, Axis is out there now and can be used to experiment a bit. So let's take a brief look at Axis and get a feel for how it's likely to work. Later, in Chapter 11, we'll take a look at JAX-RPC.

The heart of the client is `Axis2DotNet`. It invokes the `getHeadlines` method on the .NET service that we used before.

[8] The actual name is currently being debated, but as of this writing it is still called Axis. The examples in this book are based on the Axis release dated December 5, 2001. Obviously, you can expect changes to the APIs as things develop.

```
package javasoap.book.ch9.clients;
import org.apache.axis.AxisFault;
import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
import org.apache.axis.client.Transport;
import org.apache.axis.encoding.XMLType;
import org.apache.axis.transport.http.HTTPConstants;
import org.apache.axis.utils.Options;
import java.net.URL;
import java.util.*;
public class Axis2DotNet {
   public static void main(String args[]) {
      try {
         URL url =
          new URL("http://mindstrm.com:8199/CorpDataService/Proxy.asmx");
         Service  service = new Service(  );
         Call call = (Call) service.createCall(  );
         call.setTargetEndpointAddress( url );
         call.setOperationName("getHeadlines");
         call.setProperty(Call.NAMESPACE,
                  "http://mindstrm.com/CorpDataService");
         call.setProperty(HTTPConstants.MC_HTTP_SOAPACTION,
              "http://mindstrm.com/CorpDataService/getHeadlines");
         call.addParameter("symbol", XMLType.XSD_STRING,
                              Call.PARAM_MODE_IN);
         call.setReturnType(XMLType.SOAP_ARRAY);
         ArrayList ret = (ArrayList)call.invoke(new Object[] {args[0]});
         Object[] res = ret.toArray(  );
         int cnt = res.length;
         for (int i = 0; i < cnt; i++) {
            System.out.println((String)res[i]);
         }
      }
      catch (Exception e) {
         System.out.println(e);
      }
   }
}
```

Axis appears to be headed toward a more structured API, which is a welcome change from the Apache SOAP implementation. That's not a knock on the work in Apache SOAP; it's a nice achievement, and it certainly works. The smarter object-oriented API emerging in Axis couldn't have been done without the lessons learned from building Apache SOAP.

So let's take a quick look at the code. We start by creating an instance of `java.net.URL`, just like we do in Apache SOAP examples. The next step is to create an instance of `org.apache.axis.client.Service` , which acts as a factory for creating `org.apache.axis.client.Call` objects, an instance of which is created by calling `service.createCall( )`. From this point on, everything is done via the `Call` object. We pass the URL to `call.setTargetEndpointAddress( )` and set the method name using `call.setOperationName( )`. The next step is to set the namespace. We determined in a previous example that the namespace we want to use is `http://mindstrm.com/CorpDataService`. The namespace is set as a property of the `Call` object by using the `call.setProperty( )` method. The namespace property is specified in the first parameter by passing `Call.NAMESPACE`, and the appropriate namespace is passed as a string in the second parameter. We also need to set the `SOAPAction` property in the same way, this time using the `Call.MC_HTTP_SOAPACTION` constant as the property name, and `http://mindstrm.com/CorpDataService/getHeadlines` as the value. Remember, .NET services require you to specify a `SOAPAction` value that corresponds to the service method.

The next step is to set up the parameters of the `Call`, using `call.addParameter( )`. The first parameter of this method is the name of the parameter to be passed to the service. This name is used as the element name in the SOAP envelope. The second parameter specifies the data type; in this case the constant `XMLType.XSD_STRING` is used to indicate that the parameter is an instance of the built-in string type. The last parameter, `Call.PARAM_MODE_IN`, indicates that the parameter is an `in` parameter. (Axis supports `out` and `in/out` parameters too.) It's interesting that the values of the parameters aren't specified in the `call.addParameter( )` method. Instead, an `Object[]` is passed to the subsequent `call.invoke( )` method, where each entry in the object array corresponds to the value of the parameter already set up using `call.addParameter( )`. We have only one parameter: the string found in `args[0]`, which contains the stock symbol. So we use `args[0]` to create an instance of `Object[]` with just one entry.

The return value of the `call.invoke( )` method is an instance of `java.util.ArrayList` that contains the headlines returned for the stock symbol passed to the service method. Calling the `toArray( )` method of the returned `ArrayList` returns an `Object[]` in which each entry is a `java.lang.String` containing a headline. We iterate over that array and write the headline to the console. I ran `Axis2DotNet` and passed GM on the command line. Here is the result I got:

```
Autos: Year of Big Sales, Little Profits
Autos: Year of Zero Financing and Profits
Report: EchoStar May Invest in Spaceway
EchoStar may invest $1 billion in Spaceway-WSJ
GM-Daewoo Motor deal due by Jan 20 - creditor bank
In Detroit, a New Definition of 'Quality'
GM's bright idea; Ford's e-commerce plans take root
GM invests in Lutz's Cunningham Motors
December U.S. Auto Sales Seen Up Over '00
RESEARCH ALERT-UBS Warburg ups GM EPS view
```

It seems pretty easy to write Axis clients of .NET services. Axis can also use a services WSDL on the fly, but we won't cover that here.

Let's modify the code to access the GLUE-based service named `urn:CorpDataServices` running at *http://mindstrm.com:8004/glue*. The new class name is `Axis2Glue`. We need to make only one modification to the `Axis2DotNet` class to make it work with our GLUE service: the URL needs to be changed to *http://tokyo:8004/glue/urn:CorpDataServices*. Everything else can remain the same. Now that's promising! Apparently the Axis team has worked toward normalizing the API, and Axis may be a very interesting implementation when it matures.